SYSTEM AND METHOD FOR COORDINATION-CENTRIC DESIGN OF SOFTWARE SYSTEMS

Related Applications

This application is a continuation of U.S. Provisional Application No. 60/213,496 filed June 23, 2000, incorporated herein by reference.

Technical Field

The present invention relates to a system and method for designing software systems using reusable software elements and communication protocols.

Background of the Invention

A system design and programming methodology is most effective when it is closely integrated and coheres tightly with its corresponding debugging techniques. In distributed and embedded system methodologies, the relationship between debugging approaches and design methodologies has traditionally been one-sided in favor of the design and programming methodologies. Design and programming methodologies are typically developed without any consideration for the debugging techniques that will later be applied to software systems designed using that design and programming methodology. While these typical debugging approaches attempt to exploit features

provided by the design and programming methodologies, the debugging techniques will normally have little or no impact on what the design and programming features are in the first place. This lack of input from debugging approaches to design and programming methodologies serves to maintain the role of debugging as an afterthought, even though in a typical system design, debugging consumes a majority of the design time. The need remains for a design and programming methodology that reflects input from, and consideration of, potential debugging approaches in order to enhance the design and reduce the implementation time of software systems.

1. <u>Packaging of Software Elements</u>

Packaging refers to the set of interfaces a software element presents to other elements in a system. Software packaging has many forms in modern methodologies. Some examples are programming language procedure call interfaces (as with libraries), TCP/IP socket interfaces with scripting languages (as with mail and Web servers), and file formats. Several typical prior art packaging styles are described below, beginning with packaging techniques used in object-oriented programming languages and continuing with a description of more generalized approaches to packaging.

A. <u>Object-Oriented Approaches to Packaging</u>

One common packaging style is based on object-oriented programming languages and provides procedure-based (method-based) packaging for software elements (objects within this framework). These procedure-based packages allow polymorphism (in which several types of objects can have identical interfaces) through subtyping, and code sharing through inheritance (deriving a new class of objects from an already existing class of objects). In a typical object-oriented programming language, an object's interface is defined by the object's methods.

Object-oriented approaches are useful in designing concurrent systems (systems with task level parallelism and multiple processing resources?) because of the availability of active objects (objects with a thread of control). Some common,

concurrent object-oriented approaches are shown in actor languages and in concurrent Eiffel.

Early object-oriented approaches featured anonymity of objects through dynamic typechecking. This anonymity of objects meant that a first object did not need to know anything about a second object in order to send a message to the second object. One unfortunate result of this anonymity of objects was that the second object could unexpectedly respond to the first object that the sent message was not understood, resulting in a lack of predictability, due to this disruption of system executions, for systems designed with this object-oriented approach.

Most modern object-oriented approaches opt to sacrifice the benefits flowing from anonymity of objects in order to facilitate stronger static typing (checking to ensure that objects will properly communicate with one another before actually executing the software system). The main result of stronger static typing is improved system predictability. However, an unfortunate result of sacrificing the anonymity of objects is a tighter coupling between those objects, whereby each object must explicitly classify, and include knowledge about, other objects to which it sends messages. In modern object-oriented approaches the package (interface) has become indistinguishable from the object and the system in which the object is a part.

The need remains for a design and programming methodology that combines the benefits of anonymity for the software elements with the benefits derived from strong static typing of system designs.

B. <u>Other Approaches to Packaging</u>

Other packaging approaches provide higher degrees of separation between software elements and their respective packages than does the packaging in object-oriented systems. For example, the packages in event-based frameworks are interfaces with ports for transmitting and receiving events. These provide loose coupling for interelement communication. However, in an event-based framework, a software designer must explicitly implement interelement state coherence between software elements as communication between those software elements. This means

that a programmer must perform the error-prone task of designing, optimizing, implementing, and debugging a specialized communication protocol for each state coherence requirement in a particular software system.

The common object request broker architecture (CORBA) provides an interface description language (IDL) for building packages around software elements written in a variety of languages. These packages are remote procedure call (RPC) based and provide no support for coordinating state between elements. With flexible packaging, an element's package is implemented as a set of co-routines that can be adapted for use with applications through use of adapters with interfaces complementary to the interface for the software element. These adapters can be application-specific—used only when the elements are composed into a system.

The use of co-routines lets a designer specify transactions or sequences of events as part of an interface, rather than just as atomic events. Unfortunately, co-routines must be executed in lock-step, meaning a transition in one routine corresponds to a transition in the other co-routine. If there is an error in one or if an expected event is lost, the interface will fail because its context will be incorrect to recover from the lost event and the co-routines will be out of sync.

The need remains for a design and programming methodology that provides software packaging that supports the implementation of state coherence in distributed concurrent systems without packaging or interface failure when an error or an unexpected event occurs.

2. Approaches to Coordination

Coordination, within the context of this application, means the predetermined ways through which software components interact. In a broader sense, coordination refers to a methodology for composing concurrent components into a complete system. This use of the term coordination differs slightly from the use of the term in the parallelizing compiler literature, in which coordination refers to a technique for maintaining programwide semantics for a sequential program decomposed into parallel subprograms.

A. Coordination Languages

Coordination languages are usually a class of tuple-space programming languages, such as Linda. A tuple is a data object containing two or more types of data that are identified by their tags and parameter lists. In tuple-space languages, coordination occurs through the use of tuple spaces, which are global multisets of tagged tuples stored in shared memory. Tuple-space languages extend existing programming languages by adding six operators: out, in, read, eval, inp, and readp. The out, in, and read operators place, fetch and remove, and fetch without removing tuples from tuple space. Each of these three operators blocks until its operation is complete. The out operator creates tuples containing a tag and several arguments. Procedure calls can be included in the arguments, but since out blocks, the calls must be performed and the results stored in the tuple before the operator can return.

The operators eval, inp, and readp are nonblocking versions of out, in, and read, respectively. They increase the expressive power of tuple-space languages. Consider the case of eval, the nonblocking version of out. Instead of evaluating all arguments of the tuple before returning, it spawns a thread to evaluate them, creating, in effect, an active tuple (whereas tuples created by out are passive). As with out, when the computation is finished, the results are stored in a passive tuple and left in tuple space. Unlike out, however, the eval call returns immediately, so that several active tuples can be left outstanding.

Tuple-space coordination can be used in concise implementations of many common interaction protocols. Unfortunately, tuple-space languages do not separate coordination issues from programming issues. Consider the annotated Linda implementation of RPC in Listing 1.

Listing 1: Linda used to emulate RPC:

```
rpcCall(args) {                                        /* C */
      out("RPCToServer", "Client", args...);
      in("Client, "ReturnFromServer", &returnValue);
      return returnValue;                              /* C */
}                                                      /* C */
Server:
```

```
...
while(true) {                                          /* C */
     in("RPCToServer", &returnAddress, args...);
     returnValue = functionCall(args);                 /* C */
     out(returnAddress, "ReturnFromServer", returnValue);
}                                                      /* C */
```

Although the implementation depicted in Listing 1 is a compact representation of an RPC protocol, the implementation still depends heavily on an accompanying programming language (in this case, C). This dependency prevents designers from creating a new Linda RPC operator for arbitrary applications of RPC. Therefore, every time a designer uses Linda for RPC, they must copy the source code for RPC or make a C-macro. This causes tight coupling, because the client must know the name of the RPC server. If the server name is passed in as a parameter, flexibility increases; however, this requires a binding phase in which the name is obtained and applied outside of the Linda framework.

The need remains for a design and programming methodology that allows implementation of communication protocols without tight coupling between the protocol implementation and the software elements with which the protocol implementation works.

A tuple space can require large quantities of dynamically allocated memory. However, most systems, and especially embedded systems, must operate within predictable and sometimes small memory requirements. Tuple-space systems are usually not suitable for coordination in systems that must operate within small predictable memory requirements because once a tuple has been generated, it remains in tuple space until it is explicitly removed or the software element that created it terminates. Maintaining a global tuple space can be very expensive in terms of overall system performance. Although much work has gone into improving the efficiency of tuple-space languages, system performance remains worse with tuple-space languages than with message-passing techniques.

The need remains for a design and programming methodology that can effectively coordinate between software elements while respecting performance and predictable memory requirements.

B. Fixed Coordination Models

In tuple-space languages, much of the complexity of coordination remains entangled with the functionality of computational elements. An encapsulating coordination formalism decouples intercomponent interactions from the computational elements.

This type of formalism can be provided by fixed coordination models in which the coordination style is embodied in an entity and separated from computational concerns. Synchronous coordination models coordinate activity through relative schedules. Typically, these approaches require the coordination protocol to be manually constructed in advance. In addition, computational elements must be tailored to the coordination style used for a particular system (which may require intrusive modification of the software elements).

The need remains for a design and programming methodology that allows for coordination between software elements without tailoring the software elements to the specific coordination style used in a particular software system.

Summary of the Invention

The present invention provides a coordination-centric design and programming methodology to facilitate both the design and the debugging of software systems. This approach includes an encapsulating formalism for coordination. In accordance with the present invention, coordination protocols are embodied in coordinators. Coordinators serve to expose the workings of a coordination protocol and organize relevant system information in a meaningful manner. This facilitates coordination-based debugging by letting designers focus on coordination as an entity separate from computation and by showing designers what is happening internally within a software system.

In accordance with the present invention, functional blocks, or software elements, are represented as components. Components contain modes that define behaviors, actions that perform these behaviors, and coordination interfaces that connect components to other components through coordinators. Packaging software elements in this fashion provides more modularity for the components than prior art design and programming methodologies. Coordination interfaces make both the data and control aspects of intercomponent interactions explicit, so that the control and data interactions can be adapted to a variety of interaction protocols without needing internal modifications to the components. The same coordination interface type can be used by a variety of components. Any component in a software system can be replaced by a new component having completely different functionality without any other components within the system needing to know that a change has been made, as long as the new component has a coordination interface appropriate for the coordinator in the software system.

Both components and coordinators can be composed hierarchically to form higher-order functionality and interaction protocols. This facilitates design complexity management. All pieces of the coordination-centric design methodology work together to provide both the software designers and the debugging tools with information about the software system's behavior and the behavior of the software system's components. This information serves to simplify software system design and software system debugging. The coordination interfaces expose control state and message traffic to scrutiny by software designers and debugging tools. Meanwhile, coordinators expose interaction protocols to scrutiny by software designers and debugging tools.

In accordance with the present invention, the coordination-centric design methodology treats design and debugging as interrelated issues and, as a result, makes complex embedded software more debuggable and more easily designable. Debugging in fact dominates embedded system software development time, and the debugging approach of the present invention facilitates the debugging of complex embedded systems and makes other aspects of the design flow, such as managing

design complexity and reusing software, easier than in prior art design and programming methodologies.

Additional aspects and advantages of this invention will be apparent from the following detailed description of preferred embodiments thereof, which proceeds with reference to the accompanying drawings.

Brief Description of the Drawings

Fig. 1 is a component in accordance with the present invention.

Fig. 2 is the component of Fig. 1 further having a set of coordination interfaces.

Fig. 3A is a prior art round-robin resource allocation protocol with a centralized controller.

Fig. 3B is a prior art round-robin resource allocation protocol implementing a token passing scheme.

Fig. 4A is a detailed view of a component and a coordination interface connected to the component for use in round-robin resource allocation in accordance with the present invention.

Fig. 4B depicts a round-robin coordinator in accordance with the present invention.

Fig. 5 shows several typical ports for use in a coordination interface in accordance with the present invention.

Fig. 6A is a unidirectional data transfer coordinator in accordance with the present invention.

Fig. 6B is a bidirectional data transfer coordinator in accordance with the present invention.

Fig. 6C is a state unification coordinator in accordance with the present invention.

Fig. 6D is a control state mutex coordinator in accordance with the present invention.

Fig. 7 is a system for implementing subsumption resource allocation having components, a shared resource, and a subsumption coordinator.

Fig. 8 is a barrier synchronization coordinator in accordance with the present invention.

Fig. 9 is a rendezvous coordinator in accordance with the present invention.

Fig. 10 depicts a dedicated RPC system having a client, a server, and a dedicated RPC coordinator coordinating the activities of the client and the server.

Fig. 11 is a compound coordinator with both preemption and round-robin coordination for controlling the access of a set of components to a shared resource.

Fig. 12A is software system with two data transfer coordinators, each having constant message consumption and generation rules and each connected to a separate data-generating component and connected to the same data-receiving component.

Fig. 12B is the software system of Fig. 12A in which the two data transfer coordinators have been replaced with a merged data transfer coordinator.

Fig. 13 is a system implementing a first come, first served resource allocation protocol in accordance with the present invention.

Fig. 14 is a system implementing a multiclient RPC coordination protocol formed by combining the first come, first served protocol of Fig. 13 with the dedicated RPC coordinator of Fig. 10.

Fig. 15 depicts a large system in which the coordination-centric design methodology can be employed having a wireless device interacting with a cellular network.

Fig. 16 shows a top-level view of the behavior and components for a system for a cell phone.

Fig. 17A is a detailed view of a GUI component of the cell phone of Fig. 16.

Fig. 17B is a detailed view of a call log component of the cell phone of Fig. 16.

Fig. 18A is a detailed view of a voice subsystem component of the cell phone of Fig. 16.

Fig. 18B is a detailed view of a connection component of the cell phone of Fig. 16.

Fig. 19 depicts the coordination layers between a wireless device and a base station, and between the base station and a switching center, of Fig. 15.

Fig. 20 depicts a cell phone call management component, a master switching center call management component, and a call management coordinator connecting the respective call management components.

Fig. 21A is a detailed view of a transport component of the connection component of Fig. 18B.

Fig. 21B is a CDMA data modulator of the transport component of Fig. 18B.

Fig. 22 is a detailed view of a typical TDMA and a typical CDMA signal for the cell phone of Fig. 16.

Fig. 23A is a LCD touch screen component for a Web browser GUI for a wireless device.

Fig. 23B is a Web page formatter component for the Web browser GUI for the wireless device.

Fig. 24A is a completed GUI system for a handheld Web browser.

Fig. 24B shows the GUI system for the handheld Web browser combined with the connection subsystem of Fig. 18B in order to access the cellular network of Fig. 15.

Detailed Description of Preferred Embodiment

Fig. 1 is an example of a component 100, which is the basic software element within the coordination-centric design framework, in accordance with the present

invention. With reference to Fig. 1, component 100 contains a set of modes 102. Each mode 102 corresponds to a specific behavior associated with component 100. Each mode 102 can either be active or inactive, respectively enabling or disabling the behavior corresponding to that mode 102. Modes 102 can make the conditional aspects of the behavior of component 100 explicit. The behavior of component 100 is encapsulated in a set of actions 104, which are discrete, event-triggered behavioral elements within the coordination-centric design methodology. Component 100 can be copied and the copies of component 100 can be modified, providing the code-sharing benefits of inheritance.

Actions 104 are enabled and disabled by modes 102, and hence can be thought of as effectively being properties of modes 102. An event (not shown) is an instantaneous condition, such as a timer tick, a data departure or arrival, or a mode change. Actions 104 can activate and deactivate modes 102, thereby selecting the future behavior of component 100. This is similar to actor languages, in which methods are allowed to replace an object's behavior.

In coordination-centric design, however, all possible behaviors must be identified and encapsulated before runtime. For example, a designer building a user interface component for a cell phone might define one mode for looking up numbers in an address book (in which the user interface behavior is to display complete address book entries in formatted text) and another mode for displaying the status of the phone (in which the user interface behavior is to graphically display the signal power and the battery levels of the phone). The designer must define both the modes and the actions for the given behaviors well before the component can be executed.

Fig. 2 is component 100 further including a first coordination interface 200, a second coordination interface 202, and a third coordination interface 204. Coordination-centric design's components 100 provide the code-sharing capability of object-oriented inheritance through copying. Another aspect of object-oriented inheritance is polymorphism through shared interfaces. In object-oriented languages, an object's interface is defined by its methods. Although coordination-centric design's actions 104 are similar to methods in object-oriented languages, they do not

define the interface for component 100. Components interact through explicit and separate coordination interfaces, in this figure coordination interfaces 200, 202, and 204. The shape of coordination interfaces 200, 202, and 204 determines the ways in which component 100 may be connected within a software system. The way coordination interfaces 200, 202, and 204 are connected to modes 102 and actions 104 within component 100 determines how the behavior of component 100 can be managed within a system. Systemwide behavior is managed through coordinators (see Fig. 4B and subsequent).

For our approach to be effective, several factors in the design of software elements must coincide: packaging, internal organization, and how elements coordinate their behavior. Although these are often treated as independent issues, conflicts among them can exacerbate debugging. We handle them in a unified framework that separates the internal activity from the external relationship of component 100. This lets designers build more modular components and encourages them to specify distributable versions of coordination protocols. Components can be reused in a variety of contexts, both distributed, and single processor 1.

1. Introduction to Coordination

Within this application, coordination refers to the predetermined ways by which components interact. Consider a common coordination activity: resource allocation. One simple protocol for this is round-robin: participants are lined up, and the resource is given to each participant in turn. After the last participant is served, the resource is given back to the first. There is a resource-scheduling period during which each participant gets the resource exactly once, whether or not it is needed.

Fig. 3A is prior art round-robin resource allocation protocol with a centralized controller 300, which keeps track of and distributes the shared resource (not shown) to each of software elements 302, 304, 306, 308, and 310 in turn. With reference to Fig. 3A, controller 300 alone determines which software element 302, 304, 306, 308, or 310 is currently allowed to use the resource and which has it next. This implementation of a round-robin protocol permits software elements 302, 304, 306, 308, and 310 to be modular, because only controller 300 keeps track of the software

elements. Unfortunately, when this implementation is implemented on a distributed architecture (not shown), controller 300 must typically be placed on a single processing element (not shown). As a result, all coordination requests must go through that processing element, which can cause a communication performance bottleneck. For example, consider the situation in which software elements 304 and 306 are implemented on a first processing element (not shown) and controller 300 is implemented on a second processing element. Software element 304 releases the shared resource and must send a message indicating this to controller 300. Controller 300 must then send a message to software element 306 to inform software element 306 that it now has the right to the shared resource. If the communication channel between the first processing resource and the second processing resource is in use or the second processing element is busy, then the shared resource must remain idle, even though both the current resource holder and the next resource holder (software elements 304 and 306 respectively) are implemented on the first processing element (not shown). The shared resource must typically remain idle until communication can take place and controller 300 can respond. This is an inefficient way to control access to a shared resource.

Fig. 3B is a prior art round-robin resource allocation protocol implementing a token passing scheme. With reference to Fig. 3B, this system consists of a shared resource 311 and a set of software elements 312, 314, 316, 318, 320, and 322. In this system a logical token 324 symbolizes the right to access resource 311, *i.e.,* when a software element holds token 324, it has the right to access resource 311. When one of software elements 312, 314, 316, 318, 320, or 322 finishes with resource 311, it passes token 324, and with token 324 the access right, to a successor. This implementation can be distributed without a centralized controller, but as shown in Figure v3B, this is less modular, because it requires each software element in the set to keep track of a successor.

Not only must software elements 312, 314, 316, 318, 320, and 322 keep track of successors, but each must implement a potentially complicated and error-prone protocol for transferring token 324 to its successor. Bugs can cause token 324 to be

lost or introduce multiple tokens 324. Since there is no formal connection between the physical system and complete topology maps (diagrams that show how each software element is connected to others within the system), some software elements might erroneously be serviced more than once per cycle, while others are completely neglected. However, these bugs can be extremely difficult to track after the system is completed. The protocol is entangled with the functionality of each software element, and it is difficult to separate the two for debugging purposes. Furthermore, if a few of the software elements are located on the same machine, performance of the implementation can be poor. The entangling of computation and coordination requires intrusive modification to optimize the system.

2. Coordination-Centric Design's Approach to Coordination

The coordination-centric design methodology provides an encapsulating formalism for coordination. Components such as component 100 interact using coordination interfaces, such as first, second, and third coordination interfaces 200, 202, and 204, respectively. Coordination interfaces preserve component modularity while exposing any parts of a component that participate in coordination. This technique of connecting components provides polymorphism in a similar fashion to subtyping in object-oriented languages.

Fig. 4A is a detailed view of a component 400 and a resource access coordination interface 402 connected to component 400 for use in a round-robin coordination protocol in accordance with the present invention. With reference to Fig. 4A, resource access coordination interface 402 facilitates implementation of a round-robin protocol that is similar to the token-passing round-robin protocol described above. Resource access coordination interface 402 has a single bit of control state, called access, which is shown as an arbitrated control port 404 that indicates whether or not component 400 is holding a virtual token (not shown). Component 400 can only use a send message port 406 on access coordination interface 402 when arbitrated control port 404 is true. Access coordination interface 402 further has a receive message port 408.

Fig. 4B show a round-robin coordinator 410 in accordance with the present invention. With reference to Fig. 4B, round-robin coordinator 410 has a set of coordinator coordination interfaces 412 for connecting to a set of components 400. Each component 400 includes a resource access coordination interface 402. Each coordinator coordination interface 412 has a coordinator arbitrated control port 414, an incoming send message port 416 and an outgoing receive message port 418. Coordinator coordination interface 412 in complimentary to resource access coordination interface 402, and vice versa, because the ports on the two interfaces are compatible and can function to transfer information between the two interfaces.

The round-robin protocol requires round-robin coordinator 410 to manage the coordination topology. Round-robin coordinator 410 is an instance of more general abstractions called coordination classes, in which coordination classes define specific coordination protocols and a coordinator is a specific implementation of the coordination class. Round-robin coordinator 410 contains all information about how components 400 are supposed to coordinate. Although round-robin coordinator 410 can have a distributed implementation, no component 400 is required to keep references to any other component 400 (unlike the distributed round-robin implementation shown in Fig. 3B). All required references are maintained by round-robin coordinator 410 itself, and components 400 do not even need to know that they are coordinating through round-robin. Resource access coordination interface 402 can be used with any coordinator that provides the appropriate complementary interface. A coordinator's design is independent of whether it is implemented on a distributed platform or on a monolithic single processor platform.

3. Coordination Interfaces

Coordination interfaces are used to connect components to coordinators. They are also the principle key to a variety of useful runtime debugging techniques. Coordination interfaces support component modularity by exposing all parts of the component that participate in the coordination protocol. Ports are elements of coordination interfaces, as are guarantees and requirements, each of which will be described in turn.

A. Ports

A port is a primitive connection point for interconnecting components. Each port is a five-tuple (T; A; Q; D; R) in which:

- T represents the data type of the port. T can be one of int, boolean, char, byte, float, double, or cluster, in which cluster represents a cluster of data types (*e.g.*, an int followed by a float followed by two bytes).
- A is a boolean value that is true if the port is arbitrated and false otherwise.
- Q is an integer greater than zero that represents logical queue depth for a port.
- D is one of in, out, inout, or custom and represents the direction data flows with respect to the port.
- R is one of discard-on-read, discard-on-transfer, or hold and represents the policy for data removal on the port. Discard-on-read indicates that data is removed immediately after it is read (and any data in the logical queue are shifted), discard-on-transfer indicates that data is removed from a port immediately after being transferred to another port, and hold indicates that data should be held until it is overwritten by another value. Hold is subject to arbitration.

Custom directionality allows designers to specify ports that accept or generate only certain specific values. For example, a designer may want a port that allows other components to activate, but not deactivate, a mode. While many combinations of port attributes are possible, we normally encounter only a few. The three most common are message ports (output or input), state ports (output, input, or both; sometimes arbitrated), and control ports (a type of state port). Fig. 5 illustrates the visual syntax used for several common ports throughout this application. With reference to Fig. 5, this figure depicts an exported state port 502, an imported state port 504, an arbitrated state port 506, an output data port 508, and an input data port 510.

1. Message Ports

Message ports (output and input) data ports 508 and 510 respectively) are either send (T; false; 1; out; discard-on-transfer) or receive (T; false; Q; in; discard-on-read). Their function is to transfer data between components. Data passed to a send port is transferred immediately to the corresponding receive port, thus it cannot be retrieved from the send port later. Receive data ports can have queues of various depths. Data arrivals on these ports are frequently used to trigger and pass data parameters into actions. Values remain on receive ports until they are read.

2. State Ports

State ports take one of three forms:

1. (T; false; 1; out; hold)
2. (T; false; 1; in; hold)
3. (T; true; 1; inout; hold)

State ports, such as exported state port 502, imported state port 504, and arbitrated state port 506, hold persistent values, and the value assigned to a state port may be arbitrated. This means that, unlike message ports, values remain on the state ports until changed. When multiple software elements simultaneously attempt to alter the value of arbitrated state port 506, the final value is determined based on arbitration rules provided by the designer through an arbitration coordinator (not shown).

State ports transfer variable values between scopes, as explained below. In coordination-centric design, all variables referenced by a component are local to that component, and these variables must be explicitly declared in the component's scope. Variables can, however, be bound to state ports that are connected to other components. In this way a variable value can be transferred between components and the variable value achieves the system-level effect of a multivariable.

3. Control Ports

Control ports are similar to state ports, but a control port is limited to having the boolean data type. Control ports are typically bound to modes. Actions interact

with a control port indirectly, by setting and responding to the values of a mode that is bound to the control port.

For example, arbitrated control port 404 shown in Fig. 4A is a control port that can be bound to a mode (not shown) containing all actions that send data on a shared channel. When arbitrated control port 404 is false, the mode is inactive, disabling all actions that send data on the channel.

B. Guarantees

Guarantees are formal declarations of invariant properties of a coordination interface. There can be several types of guarantees, such as timing guarantees between events, guarantees between control state (*e.g.,* state A and state B are guaranteed to be mutually exclusive), etc. Although a coordination interface's guarantees reflect properties of the component to which the coordination interface is connected, the guarantees are not physically bound to any internal portions of the component. Guarantees can often be certified through static analysis of the software system. Guarantees are meant to cache various properties that are inherent in a component or a coordinator in order to simplify static analysis of the software system.

A guarantee is a promise provided by a coordination interface. The guarantee takes the form of a predicate promised to be invariant. In principle, guarantees can include any type of predicate (*e.g.,* $x > 3$, in which x is an integer valued state port, or $t_{ea} - t_{eb} < 2ms$). Throughout the remainder of this application, guarantees will be only event-ordering guarantees (guarantees that specify acceptable orders of events) or control-relationship guarantees (guarantees pertaining to acceptable relative component behaviors).

C. Requirements

A requirement is a formal declaration of the properties necessary for correct software system functionality. An example of a requirement is a required response time for a coordination interface—the number of messages that must have arrived at the coordination interface before the coordination interface can transmit, or fire, the messages. When two coordination interfaces are bound together, the requirements of the first coordination interface must be conservatively matched by the guarantees of

the second coordination interface (*e.g.*, $x < 7$ as a guarantee conservatively matches $x < 8$ as a requirement). As with guarantees, requirements are not physically bound to anything within the component itself. Guarantees can often be verified to be sufficient for the correct operation of the software system in which the component is used. In sum, a requirement is a predicate on a first coordination interface that must be conservatively matched with a guarantee on a complementary second coordination interface.

D. Conclusion Regarding Coordination Interfaces

A coordination interface is a four-tuple (P; G; R; I) in which:

- P is a set of named ports.
- G is a set of named guarantees provided by the interface.
- R is a set of named requirements that must be matched by guarantees of connected interfaces.
- I is a set of named coordination interfaces.

As this definition shows, coordination interfaces are recursive. Coordinator coordination interface 412, shown in Fig. 4B, used for round-robin coordination is called *AccessInterface* and is defined in Table 1.

Constituent	**Value**
ports	$P = \{$ access:StatePort, s:outMessagePort, r:inMessagePort $\}$
guarantees	$G = \{ \neg \text{access} \Rightarrow \neg \text{s.gen} \}$
requirements	$R = \varnothing$
interfaces	$I = \varnothing$

Related to coordination interfaces is a recursive coordination interface descriptor, which is a five-tuple (P_a; G_a; R_a; I_d; N_d) in which:

- P_a is a set of abstract ports, which are ports that may be incomplete in their attributes (*i.e.*, they do not yet have a datatype).

- G_a is a set of abstract guarantees, which are guarantees between abstract ports.
- R_a is a set of abstract requirements, which are requirements between abstract ports.
- I_d is a set of coordination interface descriptors.
- N_d is an element of Q x Q, where $Q = \{\infty\} \cup Z+$ and $Z+$ denotes the set of positive integers. N_d indicates the number or range of numbers of permissible interfaces (*e.g.*, [2], [2, 30], etc.).

Allowing coordination interfaces to contain other coordination interfaces is a powerful feature. It lets designers use common coordination interfaces as complex ports within other coordination interfaces. For example, the basic message ports described above are nonblocking, but we can build a blocking coordination interface (not shown) that serves as a blocking port by combining a wait state port with a message port.

4. Coordinators

A coordinator provides the concrete representations of intercomponent aspects of a coordination protocol. Coordinators allow a variety of static analysis debugging methodologies for software systems created with the coordination-centric design methodology. A coordinator contains a set of coordination interfaces and defines the relationships the coordination interfaces. The coordination interfaces complement the component coordination interfaces provided by components operating within the protocol. Through matched interface pairs, coordinators effectively describe connections between message ports, correlations between control states, and transactions between components.

For example, round-robin coordinator 410, shown in Fig. 4B, must ensure that only one component 400 has its component control port 404's value, or its access bit, set to true. Round-robin coordinator 410 must further ensure that the correct component 400 has its component control port 404 set to true for the chosen sequence. This section presents formal definitions of the parts that comprise

coordinators: modes, actions, bindings, action triples, and constraints. These definitions culminate in a formal definition of coordinators.

A. Modes

A mode is a boolean value that can be used as a guard on an action. In a coordinator, the mode is most often bound to a control port in a coordination interface for the coordinator. For example, in round-robin coordinator 410, the modes of concern are bound to a coordinator control port 414 of each coordinator coordination interface 412.

B. Actions

An action is a primitive behavioral element that can:

- Respond to events.
- Generate events.
- Change modes.

Actions can range in complexity from simple operations up to complicated pieces of source code. An action in a coordinator is called a transparent action because the effects of the action can be precomputed and the internals of the action are completely exposed to the coordination-centric design tools.

C. Bindings

Bindings connect input ports to output ports, control ports to modes, state ports to variables, and message ports to events. Bindings are transparent and passive. Bindings are simply conduits for event notification and data transfer. When used for event notification, bindings are called triggers.

D. Action Triples

To be executed, an action must be enabled by a mode and triggered by an event. The combination of a mode, trigger, and action is referred to as an action triple, which is a triple (m; t; a) in which:

- m is a mode.
- t is a trigger.
- a is an action.

The trigger is a reference to an event type, but it can be used to pass data into the action. Action triples are written: mode : trigger : action

A coordinator's actions are usually either pure control, in which both the trigger and action performed affect only control state, or pure data, in which both the trigger and action performed occur in the data domain. In the case of round-robin coordinator 410, the following set of actions is responsible for maintaining the appropriate state:

$$\text{access}_i \; : -\text{access}_i : \; +\text{access}_{(i+1) \bmod n}$$

The symbol "+" signifies a mode's activation edge (*i.e.*, the event associated with the mode becoming true), and the symbol "-" signifies its deactivation edge. When any coordinator coordination interface 412 deactivates its arbitrated control port 404's, access bit, the access bit of the next coordinator coordination interface 412 is automatically activated.

E. Constraints

In this dissertation, constraints are boolean relationships between control ports. They take the form:

$$\text{Condition} \Rightarrow \text{Effect}$$

This essentially means that the Condition (on the left side of the arrow) being true implies that Effect (on the right side of the arrow) is also true. In other words, if Condition is true, then Effect should also be true.

A constraint differs from a guarantee in that the guarantee is limited to communicating in-variant relationships between components without providing a way to enforce the in-variant relationship. The constraint, on the other hand, is a set of instructions to the runtime system dealing with how to enforce certain relationships between components. When a constraint is violated, two corrective actions are available to the system: (1) modify the values on the left-hand side to make the left-hand expression evaluate as false (an effect termed backpressure in [27]) or (2) alter the right-hand side to make it true. We refer to these techniques as LHM

(left-hand modify) and RHM (right-hand modify). For example, given the constraint $x \Rightarrow \neg y$ and the value $x \wedge y$, with RHM semantics the runtime system must respond by disabling y or setting y to false. Thus the value of $\neg y$ is set to true.

The decision of whether to use LHM, to use RHM, or even to suspend enforcement of a constraint in certain situations can dramatically affect the efficiency and predictability of the software system. Coordination-centric design does not attempt to solve simultaneous constraints at runtime. Rather, runtime algorithms use local ordered constraint solutions. This, however, can result in some constraints being violated and is discussed further below.

Round-robin coordinator 410 has a set of safety constraints to ensure that there is never more than one token in the system:

$$\text{access}_i \;\Rightarrow\; \forall_{j \neq i} \neg \text{access}_j$$

The above equation translates roughly as access_i implies not access_j for the set of all access_j where j is not equal to i. Even this simple constraint system can cause problems with local resolution semantics (as are LHM and RHM). If the runtime system attempted to fix all constraints simultaneously, all access modes would be shut down. If they were fixed one at a time, however, any duplicate tokens would be erased on the first pass, satisfying all other constraints and leaving a single token in the system.

Since high-level protocols can be built from combinations of lower-level protocols, coordinators can be hierarchically composed. A coordinator is a six-tuple (I; M; B; N; A; X) in which:

- I is a set of coordination interfaces.
- M is a set of modes.
- B is a set of bindings between interface elements (*e.g.*, control ports and message ports) and internal elements (*e.g.*, modes and triggers).
- N is a set of constraints between interface elements.
- A is a set of action triples for the coordinator.

- X is a set of subcoordinators.

Figs. 6A, 6B, 6C, and 6D show a few simple coordinators highlighting the bindings and constraints of the respective coordinators. With reference to Fig. 6A, a unidirectional data transfer coordinator 600 transfers data in one direction between two components (not shown) by connecting incoming receive message port 408 to outgoing receive message port 418 with a binding 602. With reference to Fig. 6B, bidirectional data transfer coordinator 604 transfers data back and forth between two components (not shown) by connecting incoming receive message port 408 to outgoing receive message port 418 with binding 602 and connecting send message port 406 to incoming send message port 416 with a second binding 602. Unidirectional data transfer coordinator 600 and bidirectional data transfer coordinator 604 simply move data from one message port to another. Thus each coordinator consists of bindings between corresponding ports on separate coordination interfaces.

With reference to Fig. 6C, state unification coordinator 606 ensures that a state port a 608 and a state port b 610 are always set to the same value. State unification coordinator 606 connects state port a 608 to state port b 610 with binding 602. With reference to Fig. 6D, control state mutex coordinator 612 has a first constraint 618 and a second constraint 620 as follows:

(1) $c \Rightarrow \neg d$ and

(2) $d \Rightarrow \neg c$.

Constraints 618 and 620 can be restated as follows:

(1) A state port c 614 having a true value implies that a state port d 616 has a false value, and

(2) State port d 616 having a true value implies that state port c 614 has a false value.

A coordinator has two types of coordination interfaces: up interfaces that connect the coordinator to a second coordinator, which is at a higher level of design hierarchy and down interfaces that connect the coordinator either to a component or to a third coordinator, which is at a lower level of design hierarchy. Down interfaces

have names preceded with "~". Round-robin coordinator 410 has six down coordination interfaces (previously referred to as coordinator coordination interface 412), with constraints that make the turning off of any coordinator control port 414 (also referred to as access control port) turn on the coordinator control port 414 of the next coordinator coordination interface 412 in line. Table 2 presents all constituents of the round-robin coordinator.

Constituent	**Value**
coordination interfaces	$I = {}^{\sim}\text{AccessInterface}_{1\text{-}6}$
modes	$M = \text{access}_{1\text{-}6}$
bindings	$B = \forall_{1\le i\le 6}({\sim}\text{AccessInterface}_i.\mathit{access},\ \mathit{access}_i) \cup$
constraints	$N = \forall_{1\le i\le 6}(\forall_{(1\le j\le 6)\wedge(i\ne j)}\ \text{access}_i \Rightarrow \neg\text{access}_j)$
actions	$A = \forall_{1\le i\le 6}\ \text{access}_i : -\text{access}_i : +\text{access}_{(i+1)}\ \text{mod}\ 6$
subcoordinators	$X = \varnothing$

This tuple describes an implementation of a round-robin coordination protocol for a particular system with six components, as shown in round-robin coordinator 410. We use a coordination class to describe a general coordination protocol that may not have a fixed number of coordinator coordination interfaces. The coordination class is a six-tuple (Ic; Mc; Bc; Nc; Ac; Xc) in which:

- Ic is a set of coordination interface descriptors in which each descriptor provides a type of coordination interface and specifies the number of such interfaces allowed within the coordination class.
- Mc is a set of abstract modes that supplies appropriate modes when a coordination class is instantiated with a fixed number of coordinator coordination interfaces.
- Bc is a set of abstract bindings that forms appropriate bindings between elements when the coordination class is instantiated.

- Nc is a set of abstract constraints that ensures appropriate constraints between coordination interface elements are in place as specified at instantiation.
- Ac is a set of abstract action triples for the coordinator.
- Xc is a set of coordination classes (hierarchy).

While a coordinator describes coordination protocol for a particular application, it requires many aspects, such as the number of coordination interfaces and datatypes, to be fixed. Coordination classes describe protocols across many applications. The use of the coordination interface descriptors instead of coordination interfaces lets coordination classes keep the number of interfaces and datatypes undetermined until a particular coordinator is instantiated. For example, a round-robin coordinator contains a fixed number of coordinator coordination interfaces with specific bindings and constraints between the message and state ports on the fixed number of coordinator coordination interfaces. A round-robin coordination class contains descriptors for the coordinator coordination interface type, without stating how many coordinator coordination interfaces, and instructions for building bindings and constraints between ports on the coordinator coordination interfaces when a particular round-robin coordinator is created.

5. <u>Components</u>

A component is a six-tuple (I; A; M; V; S; X) in which:

- I is a set of coordination interfaces.
- A is a set of action triples.
- M is a set of modes.
- V is a set of typed variables.
- S is a set of subcomponents.
- X is a set of coordinators used to connect the subcomponents to each other and to the coordination interfaces.

Actions within a coordinator are fairly regular, and hence a large number of actions can be described with a few simple expressions. However, actions within a component are frequently diverse and can require distinct definitions for each

individual action. Typically a component's action triples are represented with a table that has three columns: one for the mode, one for the trigger, and one for the action code. Table 3 shows some example actions from a component that can use round-robin coordination.

Mode	Trigger	Action
access	tick	AccessInterface.s.send("Test message"); -access;
¬ access	tick	waitCount++;

A component resembles a coordinator in several ways (for example, the modes and coordination interfaces in each are virtually the same). Components can have internal coordinators, and because of the internal coordinators, components do not always require either bindings or constraints. In the following subsections, various aspects of components are described in greater detail. Theses aspects of components include variable scope, action transparency, and execution semantics for systems of actions.

A. Variable Scope

To enhance a component's modularity, all variables accessed by an action within the component are either local to the action, local to the immediate parent component of the action, or accessed by the immediate parent component of the action via state ports in one of the parent component's coordination interfaces. For a component's variables to be available to a hierarchical child component, they must be exported by the component and then imported by the child of the component.

B. Action Transparency

An action within a component can be either a transparent action or an opaque action. Transparent and opaque actions each have different invocation semantics. The internal properties, *i.e.* control structures, variable, changes in state, operators, etc., of transparent actions are visible to all coordination-centric design tools. The design tools can separate, observe, and analyze all the internal properties of opaque

actions. Opaque actions are source code. Opaque actions must be executed directly, and looking at the internal properties of opaque actions can be accomplished only through traditional, source-level debugging techniques. An opaque action must explicitly declare any mode changes and coordination interfaces that the opaque action may directly affect.

C. Action Execution

An action is triggered by an event, such as data arriving or departing a message port, or changes in value being applied to a state port. An action can change the value of a state port, generate an event, and provide a way for the software system to interact with low-level device drivers. Since actions typically produce events, a single trigger can be propagated through a sequence of actions.

6. Protocols Implemented with Coordination Classes

In this section, we describe several coordinators that individually implement some common protocols: subsumption, barrier synchronization, rendezvous, and dedicated RPC.

A. Subsumption Protocol

A subsumption protocol is a priority-based, preemptive resource allocation protocol commonly used in building small, autonomous robots, in which the shared resource is the robot itself.

Fig. 7 shows a set of coordination interfaces and a coordinator for implementing the subsumption protocol. With reference to Fig. 7, a subsumption coordinator 700 has a set of subsumption coordinator coordination interfaces 702, which have a subsume arbitrated coordinator control port 704 and an incoming subsume message port 706. Each subsume component 708 has a subsume component coordination interface 710. Subsume component coordination interface 710 has a subsume arbitrated component control port 712 and an outgoing subsume message port 714. Subsumption coordinator 700 and each subsume component 708 are connected by their respective coordination interfaces, 702 and 710. Each subsumption coordinator coordination interface 702 in subsumption coordinator 700 is associated with a priority. Each subsume component 708 has a behavior that can be

applied to a robot (not shown). At any time, any subsume component 708 can attempt to assert its behavior on the robot. The asserted behavior coming from the subsume component 708 connected to the subsumption coordinator coordination interface 702 with the highest priority is the asserted behavior that will actually be performed by the robot. Subsume components 708 need not know anything about other components in the system. In fact, each subsume component 708 is designed to perform independently of whether their asserted behavior is performed or ignored.

Subsumption coordinator 700 further has a slave coordinator coordination interface 716, which has an outgoing slave message port 718. Outgoing slave message port 718 is connected to an incoming slave message port 720. Incoming slave message port 720 is part of a slave coordination interface 722, which is connected to a slave 730. When a subsume component 708 asserts a behavior and that component has the highest priority, subsumption coordinator 700 will control slave 730 (which typically controls the robot) based on the asserted behavior.

The following constraint describes the basis of the subsumption coordinator 700's behavior:

$$\mathrm{subsume}_p \Rightarrow \bigwedge_{i=1}^{p-1} \neg \mathrm{subsume}_i$$

This means that if any subsume component 708 has a subsume arbitrated component control port 712 that has a value of true, then all lower-priority subsume arbitrated component control ports 712 are set to false. An important difference between round-robin and subsumption is that in round-robin, the resource access right is transferred only when surrendered. Therefore, round-robin coordination has cooperative release semantics. However, in subsumption coordination, a subsume component 708 tries to obtain the resource whenever it needs to and succeeds only when it has higher priority than any other subsume component 708 that needs the resource at the same time. A lower-priority subsume component 708 already using the resource must surrender the resource whenever a higher-priority subsume

component 708 tries to access the resource. Subsumption coordination uses preemptive release semantics, whereby each subsume component 708 must always be prepared to relinquish the resource.

Table 4 presents the complete tuple for the subsumption coordinator.

Constituent	Value
coordination interfaces	$I =$ (Subsume$_{1\text{-}n}$) $\cup$ (Output)
modes	$M =$ subsume$_{1\text{-}n}$
bindings	$B = \forall_{1 \le i \le n}$ (Subsume$_i$.subsume, subsume$_i$) $\cup$
constraints	$N = \forall_{1 \le i \le n} (\forall_{(1 \le j \le i)}$ subsume$_i \Rightarrow \neg$subsume$_j$)
actions	$A = \varnothing$
subcoordinators	$X = \varnothing$

B. <u>Barrier Synchronization Protocol</u>

Other simple types of coordination that components might engage in enforce synchronization of activities. An example is barrier synchronization, in which each component reaches a synchronization point independently and waits. Fig. 8 depicts a barrier synchronization coordinator 800. With reference to Fig. 8, barrier synchronization coordinator 800 has a set of barrier synchronization coordination interfaces 802, each of which has a coordinator arbitrated state port 804, named wait. Coordinator arbitrated state port 804 is connected to a component arbitrated state port 806, which is part of a component coordination interface 808. Component coordination interface 808 is connected to a component 810. When all components 810 reach their respective synchronization points, they are all released from waiting. The actions for a barrier synchronization coordinator with n interfaces are:

$$\bigwedge_{0 \le i < n} wait_i \; : \; : \; \forall_{0 \le j < n} - wait_j$$

In other words, when all wait modes (not shown) become active, each one is released. The blank between the two colons indicates that the trigger event is the guard condition becoming true.

C. Rendezvous Protocol

A resource allocation protocol similar to barrier synchronization is called rendezvous. Fig. 9 depicts a rendezvous coordinator 900 in accordance with the present invention. With reference to Fig. 9, rendezvous coordinator 900 has a rendezvous coordination interface 902, which has a rendezvous arbitrated state port 904. A set of rendezvous components 906, each of which may perform different functions or have vastly different actions and modes, has a rendezvous component coordination interface 908, which includes a component arbitrated state port 910. Rendezvous components 906 connect to rendezvous coordinator 900 through their respective coordination interfaces, 908 and 902. Rendezvous coordinator 900 further has a rendezvous resource coordination interface 912, which has a rendezvous resource arbitrated state port 914, also called available. A resource 916 has a resource coordination interface 918, which has a resource arbitrated state port 920. Resource 916 is connected to rendezvous coordinator 900 by their complementary coordination interfaces, 918 and 912 respectively.

With rendezvous-style coordination, there are two types of participants: resource 916 and several resource users, here rendezvous components 916. When resource 916 is available, it activates its resource arbitrated state port 920, also referred to as its available control port. If there are any waiting rendezvous components 916, one will be matched with the resource; both participants are then released. This differs from subsumption and round-robin in that resource 916 plays an active role in the protocol by activating its available control port 920.

The actions for rendezvous coordinator 900 are:

$$available_i \wedge wait_j : \; : -available_i, -wait_j$$

This could also be accompanied by other modes that indicate the status after the rendezvous. With rendezvous coordination, it is important that only one component at a time be released from wait mode.

D. Dedicated RPC Protocol

A coordination class that differs from those described above is dedicated RPC. Fig. 10 depicts a dedicated RPC system. With reference to Fig. 10, a dedicated RPC coordinator 1000 has an RPC server coordination interface 1002, which includes an RPC server imported state port 1004, an RPC server output message port 1006, and an RPC server input message port 1008. Dedicated RPC coordinator 1000 is connected to a server 1010. Server 1010 has a server coordination interface 1012, which has a server exported state port 1014, a server input data port 1016, and a server output data port 1018. Dedicated RPC coordinator 1000 is connected to server 1010 through their complementary coordination interfaces, 1002 and 1012 respectively. Dedicated RPC coordinator 1000 further has an RPC client coordination interface 1020, which includes an RPC client imported state port 1022, an RPC client input message port 1024, and an RPC client output message port 1026. Dedicated RPC coordinator 1000 is connected to a client 1028 by connecting RPC client coordination interface 1020 to a complementary client coordination interface 1030. Client coordination interface 1030 has a client exported state port 1032, a client output message port 1034, and a client input message port 1036.

The dedicated RPC protocol has a client/server protocol in which server 1010 is dedicated to a single client, in this case client 1028. Unlike the resource allocation protocol examples, the temporal behavior of this protocol is the most important factor in defining it. The following transaction listing describes this temporal behavior:

Client 1028 enters blocked mode by changing the value stored at client exported state port 1032 to true.

Client 1028 transmits an argument data message to server 1010 via client output message port 1034.

Server 1010 receives the argument (labeled "a") data message via server input data port 1016 and enters serving mode by changing the value stored in server exported state port 1014 to true.

Server 1010 computes return value.

Server 1010 transmits a return (labeled "r") message to client 1020 via server output data port 1018 and exits serving mode by changing the value stored in server exported state port 1014 to false.

Client 1028 receives the return data message via client input message port 1036 and exits blocked mode by changing the value stored at client exported state port 1032 to false.

This can be presented more concisely with an expression describing causal relationships:

$$\begin{aligned} T_{RPC} = {} & +client.blocked \rightarrow client.transmits \rightarrow \\ & +server.serving \rightarrow server.transmits \rightarrow \\ & (-server.serving \parallel client.receives) \rightarrow -client.blocked \end{aligned}$$

The transactions above describe what is supposed to happen. Other properties of this protocol must be described with temporal logic predicates.

$$server.serving \Rightarrow client.blocked$$

$$server.serving \Rightarrow F(server.r.output)$$

$$server.a.input \Rightarrow F(server.serving)$$

The *r* in *server.r.output* refers to the server output data port 1018, also labeled as the *r* event port on the server, and the *a* in *serving.a.input* refers to server input data port 1016, also labeled as the *a* port on the server (see Fig. 10).

Together, these predicates indicate that (1) it is an error for server 1010 to be in serving mode if client 1028 is not blocked; (2) after server 1010 enters serving mode, a response message is sent or else an error occurs; and (3) server 1010 receiving a message means that server 1010 must enter serving mode. Relationships between control state and data paths must also be considered, such as:

$(client.a \Rightarrow client.blocked)$

In other words, client 1028 must be in blocked mode whenever it sends an argument message.

The first predicate takes the same form as a constraint; however, since dedicated RPC coordinator 1000 only imports the client:blocked and server:serving modes (*i.e.*, through RPC client imported state port 1022 and RPC server imported state port 1004 respectively), dedicated RPC coordinator 1000 is not allowed to alter these values to comply. In fact, none of these predicates is explicitly enforced by a runtime system. However, the last two can be used as requirements and guarantees for interface type-checking.

7. <u>System-Level Execution</u>

Coordination-centric design methodology lets system specifications be executed directly, according to the semantics described above. When components and coordinators are composed into higher-order structures, however, it becomes essential to consider hazards that can affect system behavior. Examples include conflicting constraints, in which local resolution semantics may either leave the system in an inconsistent state or make it cycle forever, and conflicting actions that undo one another's behavior. In the remainder of this section, the effect of composition issues on system-level executions is explained.

A. <u>System Control Configurations</u>

A configuration is the combined control state of a system—basically, the set of active modes at a point in time. In other words, a configuration in coordination-centric design is a bit vector containing one bit for each mode in the system. The bit representing a control state is true when the control state is active and false when the control state is inactive. Configurations representing the complete

system control state facilitate reasoning on system properties and enable several forms of static analysis of system behavior.

B. Action-Trigger Propagation

Triggers are formal parameters for events. As mentioned earlier, there are two types of triggers: (1) control triggers, invoked by control events such as mode change requests, and (2) data flow triggers, invoked by data events such as message arrivals or departures. Components and coordinators can both request mode changes (on the modes visible to them) and generate new messages (on the message ports visible to them). Using actions, these events can be propagated through the components and coordinators in the system, causing a cascade of data transmissions and mode change requests, some of which can cancel other requests. When the requests, and secondary requests implied by them, are all propagated through the system, any requests that have not been canceled are confirmed and made part of the system's new configuration.

Triggers can be immediately propagated through their respective actions or delayed by a scheduling step. Recall that component actions can be either transparent or opaque. Transparent actions typically propagate their triggers immediately, although it is not absolutely necessary that they do so. Opaque actions typically must always delay propagation.

1. Immediate Propagation

Some triggers must be immediately propagated through actions, but only on certain types of transparent actions. Immediate propagation can often involve static precomputation of the effect of changes, which means that certain actions may never actually be performed. For example, consider a system with a coordinator that has an action that activates mode A and a coordinator with an action that deactivates mode B whenever A is activated. Static analysis can be used to determine in advance that any event that activates A will also deactivate B; therefore, this effect can be executed immediately without actually propagating it through A.

2. Delayed Propagation

Trigger propagation through opaque actions must typically be delayed, since the system cannot look into opaque actions to precompute their results. Propagation may be delayed for other reasons, such as system efficiency. For example, immediate propagation requires tight synchronization among software components. If functionality is spread among a number of architectural components, immediate propagation is impractical.

C. A Protocol Implemented with a Compound Coordinator

Multiple coordinators are typically needed in the design of a system. The multiple coordinators can be used together for a single, unified behavior. Unfortunately, one coordinator may interfere with another's behavior.

Fig. 11 shows a combined coordinator 1100 with both preemption and round-robin coordination for controlling access to a resource, as discussed above. With reference to Fig. 11, components 1102, 1104, 1106, 1108, and 1110 primarily use round-robin coordination, and each includes a component coordination interface 1112, which has a component arbitrated control port 1114 and a component output message port 1116. However, when a preemptor component 1120 needs the resource, preemptor component 1120 is allowed to grab the resource immediately. Preemptor component 1120 has a preemptor component coordination interface 1122. Preemptor component coordination interface 1122 has a preemptor arbitrated state port 1124, a preemptor output message port 1126, and a preemptor input message port 1128.

All component coordination interfaces 1112 and preemptor component coordination interface 1122 are connected to a complementary combined coordinator coordination interface 1130, which has a coordinator arbitrated state port 1132, a coordinator input message port 1134, and a coordinator output message port 1136. Combined coordinator 1100 is a hierarchical coordinator and internally has a round-robin coordinator (not shown) and a preemption coordinator (not shown). Combined coordinator coordination interface 1130 is connected to a coordination interface to round-robin 1138 and a coordination interface to preempt 1140. Coordinator arbitrated state port 1132 is bound to both a token arbitrated control port

1142, which is part of coordination interface to round-robin 1138, and to a preempt arbitrated control port 1144, which is part of coordination interface to preempt 1140. Coordinator input message port 1134 is bound to an interface to a round-robin output message port 1146, and coordinator output message port 1136 is bound to an interface to round-robin input message port 1148.

Thus preemption interferes with the normal round-robin ordering of access to the resource. After a preemption-based access, the resource moves to the component that in round-robin-ordered access would be the successor to preemptor component 1120. If the resource is preempted too frequently, some components may starve.

D. Mixing Control and Data in Coordinators

Since triggers can be control-based, data-based, or both, and actions can produce both control and data events, control and dataflow aspects of a system are coupled through actions. Through combinations of actions, designers can effectively employ modal data flow, in which relative schedules are switched on and off based on the system configuration.

Relative scheduling is a form of coordination. Recognizing this and understanding how it affects a design can allow a powerful class of optimizations. Many data-centric systems (or subsystems) use conjunctive firing, which means that a component buffers messages until a firing rule is matched. When matching occurs, the component fires, consuming the messages in its buffer that caused it to fire and generating a message or messages of its own. Synchronous data flow systems are those in which all components have only firing rules with constant message consumption and generation.

Fig. 12A shows a system in which a component N1 1200 is connected to a component N3 1202 by a data transfer coordinator 1204 and a component N2 1206 is connected to component N3 1202 by a second data transfer coordinator 1208. Component N3 1202 fires when it accumulates three messages on a port c 1210 and two messages on a port d 1212. On firing, component N3 1202 produces two messages on a port o 1214. Coordination control state tracks the logical buffer depth

for these components. This is shown with numbers representing the logical queue depth of each port in Fig. 12.

Fig. 12B shows the system of Fig. 12A in which data transfer coordinator 1204 and second data transfer coordinator 1208 have been merged to form a merged data transfer coordinator 1216. Merging the coordinators in this example provides an efficient static schedule for component firing. Merged data transfer coordinator 1216 fires component N1 1200 three times and component N2 1206 twice. Merged data transfer coordinator 1216 then fires component N3 1202 twice (to consume all messages produced by component N1 1200 and component N2 1206).

Message rates can vary based on mode. For example, a component may consume two messages each time it fires in one mode and four each time it fires in a second mode. For a component like this, it is often possible to merge schedules on a configuration basis, in which each configuration has static consumption and production rates for all affected components.

E. Coordination Transformations

In specifying complete systems, designers must often specify not only the coordination between two objects, but also the intermediate mechanism they must use to implement this coordination. While this intermediate mechanism can be as simple as shared memory, it can also be another coordinator; hence coordination may be, and often is, layered. For example, RPC coordination often sits on top of a TCP/IP stack or on an IrDA stack, in which each layer coordinates with peer layers on other processing elements using unique coordination protocols. Here, each layer provides certain capabilities to the layer directly above it, and the upper layer must be implemented in terms of them.

In many cases, control and communication synthesis can be employed to automatically transform user-specified coordination to a selected set of standard protocols. Designers may have to manually produce transformations for nonstandard protocols.

F. Dynamic Behavior with Compound Coordinators

Even in statically bound systems, components may need to interact in a fashion that appears dynamic. For example, RPC-style coordination often has multiple clients for individual servers. Here, there is no apparent connection between client and server until one is forged for a transaction. After the connection is forged, however, the coordination proceeds in the same fashion as dedicated RPC.

Our approach to this is to treat the RPC server as a shared resource, requiring resource allocation protocols to control access. However, none of the resource allocation protocols described thus far would work efficiently under these circumstances. In the following subsections, an appropriate protocol for treating the RPC as a shared resource will be described and how that protocol should be used as part of a complete multiclient RPC coordination class—one that uses the same RPC coordination interfaces described earlier—will be discussed.

1. First Come/First Serve protocol (FCFS)

Fig. 13 illustrates a first come/first serve (FCFS) resource allocation protocol, which is a protocol that allocates a shared resource to the requester that has waited longest. With reference to Fig. 13, a FCFS component interface 1300 for this protocol has a request control port 1302, an access control port 1304 and a component outgoing message port 1306. A FCFS coordinator 1308 for this protocol has a set of FCFS interfaces 1310 that are complementary to FCFS component interfaces 1300, having a FCFS coordinator request control port 1312, a FCFS coordinator access port 1314, and a FCFS coordinator input message port 1316. When a component 1318 needs to access a resource 1320, it asserts request control port 1302. When granted access, FCFS coordinator 1308 asserts the appropriate FCFS coordinator access port 1314, releasing FCFS coordinator request control port 1312.

To do this, FCFS coordinator 1308 uses a rendezvous coordinator and two round-robin coordinators. One round-robin coordinator maintains a list of empty slots in which a component may be enqueued, and the other round-robin coordinator maintains a list showing the next component to be granted access. When an FCFS coordinator request control port 1312 becomes active, FCFS coordinator 1308 begins

a rendezvous access to a binder action. When activated, this action maps the appropriate component 1318 to a position in the round-robin queues. A separate action cycles through one of the queues and selects the next component to access the server. As much as possible, FCFS coordinator 1308 attempts to grant access to resource 1320 to the earliest component 1318 having requested resource 1320, with concurrent requests determined based on the order in the rendezvous coordinator of the respective components 1318.

2. Multiclient RPC

Fig. 14 depicts a multiclient RPC coordinator 1400 formed by combining FCFS coordinator 1308 with dedicated RPC coordinator 1000. With reference to Fig. 14, a set of clients 1402 have a set of client coordination interfaces 1030, as shown in Fig. 10. In addition, multiclient RPC coordinator 1400 has a set of RPC client coordination interfaces 1020, as shown in Fig. 10. For each RPC client coordination interface 1020, RPC client input message port 1024, of RPC client coordination interface 1020, is bound to the component outgoing message port 1306 of FCFS coordinator 1308. Message transfer action 1403 serves to transfer messages between RPC client input message port 1024 and component outgoing message port 1306. For coordinating the actions of multiple clients 1402, multiclient RPC coordinator 1400 must negotiate accesses to a server 1404 and keep track of the values returned by server 1404.

F. Monitor Modes and Continuations

Features such as blocking behavior and exceptions can be implemented in the coordination-centric design methodology with the aid of monitor modes. Monitor modes are modes that exclude all but a selected set of actions called continuations, which are actions that continue a behavior started by another action.

1. Blocking Behavior

With blocking behavior, one action releases control while entering a monitor mode, and a continuation resumes execution after the anticipated response event. Monitor mode entry must be immediate (at least locally), so that no unexpected actions can execute before they are blocked by such a mode.

Each monitor mode has a list of actions that cannot be executed when it is entered. The allowed (unlisted) actions are either irrelevant or are continuations of the action that caused entry into this mode. There are other conditions, as well. This mode requires an exception action if forced to exit. However, this exception action is not executed if the monitor mode is turned off locally.

When components are distributed over a number of processing elements, it is not practical to assume complete synchronization of the control state. In fact, there are a number of synchronization options available as detailed in Chou, P "Control Composition and Synthesis of Distributed Real-Time Embedded Systems", Ph.D. dissertation, University of Washington, 1998.

2. Exception Handling

Exception actions are a type of continuation. When in a monitor mode, exception actions respond to unexpected events or events that signal error conditions. For example, multiclient RPC coordinator 1400 can bind $\neg$*client.blocked* to a monitor mode and set an exception action on +*server.serving*. This will signal an error whenever the server begins to work when the client is not blocked for a response.

8. A Complete System Example

Figure 15 depicts a large-scale example system under the coordination-centric design methodology. With reference to Fig. 15, the large scale system is a bimodal digital cellular network 1500. Network 1500 is for the most part a simplified version of a GSM (global system for mobile communications) cellular network. This example shows in greater detail how the parts of coordination-centric design work together and demonstrates a practical application of the methodology. Network 1500 has two different types of cells, a surface cell 1502 (also referred to as a base station 1502) and a satellite cell 1504. These cells are not only differentiated by physical position, but by the technologies they use to share network 1500. Satellite cells 1504 use a code division multiple access (CDMA) technology, and surface cells 1502 use a time division multiple access (TDMA) technology. Typically, there are seven frequency bands reserved for TDMA and one band reserved for CDMA. The goal is for as

much communication as possible to be conducted through the smaller TDMA cells, here surface cells 1502, because power requirements for a CDMA cells, here satellite cell 1504, increase with the number of users in the CDMA cell. Mobile units 1506, or wireless devices, can move between surface cells 1502, requiring horizontal handoffs between surface cells 1502. Several surface cells 1502 are typically connected to a switching center 1508. Switching center 1508 is typically connected to a telephone network or the Internet 1512. In addition to handoffs between surface cells 1502, the network must be able to hand off between switching centers 1508. When mobile units 1506 leave the TDMA region, they remain covered by satellite cells 1504 via vertical handoffs between cells. Since vertical handoffs require changing protocols as well as changing base stations and switching centers, they can be complicated in terms of control.

Numerous embedded systems comprise the overall system. For example, switching center 1508 and base stations, surface cells 1502, are required as part of the network infrastructure, but cellular phones, handheld Web browsers, and other mobile units 1506 may be supported for access through network 1500. This section concentrates on the software systems for two particular mobile units 1506: a simple digital cellular phone (shown in Fig. 16) and a handheld Web browser (shown in Fig. 24). These examples require a wide variety of coordinators and reusable components. Layered coordination is a feature in each system, because a function of many subsystems is to perform a layered protocol. Furthermore, this example displays how the hierarchically constructed components can be applied in a realistic system to help manage the complexity of the overall design.

To begin this discussion, we describe the cellular phone in detail, focusing on its functional components and the formalization of their interaction protocols. We then discuss the handheld Web browser in less detail but highlight the main ways in which its functionality and coordination differ from those of the cellular phone. In describing the cellular phone, we use a top-down approach to show how a coherent system organization is preserved, even at a high level. In describing the handheld

Web browser, we use a bottom-up approach to illustrate component reuse and bottom-up design.

A. Cellular Phone

Fig. 16 shows a top-level coordination diagram of the behavior of a cell phone 1600. Rather than using a single coordinator that integrates the components under a single protocol, we use several coordinators in concert. Interactions between coordinators occur mainly within the components to which they connect.

With reference to Fig. 16, cell phone 1600 supports digital encoding of voice streams. Before it can be used, it must be authenticated with a home master switching center (not shown). This authentication occurs through a registered master switch for each phone and an authentication number from the phone itself. There are various authentication statuses, such as full access, grey-listed, or blacklisted. For cell phone 1600, real-time performance is more important than reliability. A dropped packet is not retransmitted, and a late packet is dropped since its omission degrades the signal less than its late incorporation.

Each component of cell phone 1600 is hierarchical. A GUI 1602 lets users enter phone numbers while displaying them and query an address book 1604 and a logs component 1606. Address book 1604 is a database that can map names to phone numbers and vice versa. GUI 1602 uses address book 1604 to help identify callers and to look up phone numbers to be dialed. Logs 1606 track both incoming and outgoing calls as they are dialed. A voice component 1608 digitally encodes and decodes, and compresses and decompresses, an audio signal. A connection component 1610 multiplexes, transmits, receives, and demultiplexes the radio signal and separates out the voice stream and caller identification information.

Coordination among the above components makes use of several of the coordinators discussed above. Between connection component 1610 and a clock 1612, and between logs 1606 and connection component 1610, are unidirectional data transfer coordinators 600 as described with reference to Fig. 6A. Between voice component 1608 and connection component 1610, and between GUI 1602 and connection component 1610, are bidirectional data transfer coordinators 604, as

described with reference to Fig. 6B. Between clock 1612 and GUI 1602 is a state unification coordinator 606, as described with reference to Fig. 6C. Between GUI 1602 and address book 1604 is a dedicated RPC coordinator 1000 as described with reference to Fig. 10, in which address book 1604 has client 1028 and GUI 1602 has server 1010.

There is also a custom GUI/log coordinator 1614 between logs 1606 and GUI 1602. GUI/log coordinator 1614 lets GUI 1602 transfer new logged information through an r output message port 1616 on a GUI coordination interface 1618 to an r input message port 1620 on a log coordination interface 1622. GUI/log coordinator 1614 also lets GUI 1602 choose current log entries through a pair of c output message ports 1624 on GUI coordination interface 1618 and a pair of c input message ports 1626 on log coordination interface 1622. Logs 1606 continuously display one entry each for incoming and outgoing calls.

1. <u>GUI Component</u>

Fig. 17A is a detailed view of GUI component 1602, of Fig. 16. With reference to Fig. 17A, GUI component 1602 has two inner components, a keypad 1700 and a text-based liquid crystal display 1702, as well as several functions of its own (not shown). Each time a key press occurs, it triggers an action that interprets the press, depending on the mode of the system. Numeric presses enter values into a shared dialing buffer. When a complete number is entered, the contents of this buffer are used to establish a new connection through connection component 1610. Table 5 shows the action triples for GUI 1602.

Mode	Trigger	Action
Idle	[1]—[#]	numBuffer.append(keypress.val)
	Send	radio.send(numBuffer.val) +outgoingCall
	Disconnect	Nil
	Leftarrow	AddressBook.forward() +lookupMode
	Rightarrow	log.lastcall() +outlog
LookupMode	Leftarrow	AddressBook.forward()
	Rightarrow	AddressBook.backward()

An "Addr Coord" coordinator 1704 includes an address book mode (not shown) in which arrow key presses are transformed into RPC calls.

2. Logs Component

Fig. 17B is a detailed view of logs component 1606, which tracks all incoming and outgoing calls. With reference to Fig. 17B, both GUI component 1602 and connection component 1610 must communicate with logs component 1606 through specific message ports. Those specific message ports include a transmitted number message port 1720, a received number message port 1722, a change current received message port 1724, a change current transmitted message port 1726, and two state ports 1728 and 1729 for presenting the current received and current transmitted values, respectively.

Logs component 1606 contains two identical single-log components: a send log 1730 for outgoing calls and a receive log 1740 for incoming calls. The interface of logs component 1606 is connected to the individual log components by a pair of adapter coordinators, Adap1 1750 and Adap2 1752. Adap1 1750 has an adapter receive interface 1754, which has a receive imported state port 1756 and a receive output message port 1758. Adap1 1750 further has an adapter send interface 1760,

which has a send imported state port 1762 and a send output message port 1764. Within Adap1, state port 1728 is bound to receive imported state port 1756, change current received message port 1724 is bound to receive output message port 1758, received number message port 1722 is bound to a received interface output message port 1766 on a received number coordination interface 1768, change current transmitted message port 1726 is bound to send output message port 1764, and state port 1729 is bound to Up.rc is bound to send imported state port 1762 .

3. Voice Component

Fig. 18A is a detailed view of voice component 1608 of Fig. 16. Voice component 1608 has a compression component 1800 for compressing digitized voice signals before transmission, a decompression component 1802 for decompressing received digitized voice signals, and interfaces 1804 and 1806 to analog transducers (not shown) for digitizing sound to be transmitted and for converting received transmissions into sound. Voice component 1608 is a pure data flow component containing sound generator 1808 which functions as a white-noise generator, a ring tone generator, and which has a separate port for each on sound generator interface 1810, and voice compression functionality in the form of compression component 1800 and decompression component 1802.

4. Connection Component

Fig. 18B is a detailed view of connection component 1610 of Fig. 16. With reference to Fig. 18B, connection component 1610 coordinates with voice component 1608, logs component 1606, clock 1612, and GUI 1602. In addition, connection component 1610 is responsible for coordinating the behavior of cell phone 1600 with a base station that owns the surface cell 1502 (shown in Fig. 15), a switching center 1508 (shown in Fig. 15), and all other phones (not shown) within surface cell 1502. Connection component 1610 must authenticate users, establish connections, and perform handoffs as needed—including appropriate changes in any low-level protocols (such as a switch from TDMA to CDMA).

Fig. 19 depicts a set of communication layers between connection component 1610 of cell phone 1600 and base station 1502 or switching center 1508. With

reference to Fig. 19, has several subcomponents, or lower-level components, each of which coordinates with an equivalent, or peer, layer on either base station 1502 or switching center 1508. The subcomponents of connection component 1610 include a cell phone call manager 1900, a cell phone mobility manager 1902, a cell phone radio resource manager 1904, a cell phone link protocol manager 1906, and a cell phone transport manager 1908 which is responsible for coordinating access to and transferring data through the shared airwaves TDMA and CDMA coordination. Each subcomponent will be described in detail including how each fits into the complete system.

Base station 1502 has a call management coordinator 1910, a mobility management coordinator 1912, a radio resource coordinator 1914 (BSSMAP 1915), a link protocol coordinator 1916 (SCCO 1917), and a transport coordinator 1918 (MTP 1919). Switching center 1508 has a switching center call manager 1920, a switching center mobility manager 1922, (a BSSMAP 1924, a SCCP 1926, and an MTP 1928).

a. Call Management

Fig. 20 is a detailed view of a call management layer 2000 consisting of cell phone call manager 1900, which is connected to switching center call manager 1920 by call management coordinator 1910. With reference to Fig. 20, call management layer 2000 coordinates the connection between cell phone 1600 and switching center 1508. Call management layer 2000 is responsible for dialing, paging, and talking. Call management layer 2000 is always present in cell phone 1600, though not necessarily in Internet appliances (discussed later). Cell phone call manager 1900 includes a set of modes (not shown) for call management coordination that consists of the following modes:

- Standby
- Dialing
- RingingRemote
- Ringing
- CallInProgress

Cell phone call manager 1900 has a cell phone call manager interface 2002. Cell phone call manager interface 2002 has a port corresponding to each of the above modes. The standby mode is bound to a standby exported state port 2010. The dialing mode is bound to a dialing exported state port 2012. The RingingRemote mode is bound to a RingingRemote imported state port 2014. The Ringing mode is bound to a ringing imported state port 2016. The CallInProgress mode is bound to a CallInProgress arbitrated state port 2018.

Switching center call manager 1920 includes the following modes (not shown) for call management coordination at the switching center:

- Dialing
- RingingRemote
- Paging
- CallInProgress

Switching center call manager 1920 has a switching center call manager coordination interface 2040, which includes a port for each of the above modes within switching center call manager 1920.

When cell phone 1600 requests a connection, switching center 1508 creates a new switching center call manager and establishes a call management coordinator 1910 between cell phone 1600 and switching center call manager 1920.

b. Mobility Management

A mobility management layer authenticates mobile unit 1506 or cell phone 1600. When there is a surface cell 1502 available, mobility manager 1902 contacts the switching center 1508 for surface cell 1502 and transfers a mobile unit identifier (not shown) for mobile unit 1506 to switching center 1508. Switching center 1508 then looks up a home motor switching center for mobile unit 1506 and establishes a set of permissions assigned to mobile unit 1506. This layer also acts as a conduit for the call management layer. In addition, the mobility management layer performs handoffs between base stations 1502 and switching centers 1508 based on information received from the radio resource layer.

c. Radio Resource

In the radio resource layer, radio resource manager 1904, chooses the target base station 1502 and tracks changes in frequencies, time slices, and CDMA codes. Cell phones may negotiate with up to 16 base stations simultaneously. This layer also identifies when handoffs are necessary.

d. Link Protocol

The link layer manages a connection between cell phone 1600 and base station 1502. In this layer, link protocol manager 1906 packages data for transfer to base station 1502 from cell phone 1600.

e. Transport

Fig. 21A is a detailed view of transport component 1908 of connection component 1610. Transport component 1908 has two subcomponents, a receive component 2100 for receiving data and a transmit component 2102 for transmitting data. Each of these subcomponents has two parallel data paths a CDMA path 2104 and a TDMA/FDMA path 2106 for communicating in the respective network protocols.

Fig. 21B is a detailed view of a CDMA modulator 2150, which implements a synchronous data flow data path. CDMA modulator 2150 takes the dot-product of an incoming data signal along path 2152 and a stored modulation code for cell phone 1600 along path 2154, which is a sequence of chips, which are measured time signals having a value of −1 or +1.

Transport component 1908 uses CDMA and TDMA technologies to coordinate access to a resource shared among several cell phones 1600, *i.e.,* the airwaves. Transport components 1908 supersede the FDMA technologies (*e.g.,* AM and FM) used for analog cellular phones and for radio and television broadcasts. In FDMA, a signal is encoded for transmission by modulating it with a carrier frequency. A signal is decoded by demodulation after being passed through a band pass filter to remove other carrier frequencies. Each base station 1502 has a set of frequencies—chosen to minimize interference between adjacent cells. (The area covered by a cell may be much smaller than the net range of the transmitters within it.)

TDMA, on the other hand, coordinates access to the airwaves through time slicing. Cell phone 1600 on the network is assigned a small time slice, during which it has exclusive access to the media. Outside of the small time slice, cell phone 1600 must remain silent. Decoding is performed by filtering out all signals outside of the small time slice. The control for this access must be distributed. As such, each component involved must be synchronized to observe the start and end of the small time slice at the same instant.

Most TDMA systems also employ FDMA, so that instead of sharing a single frequency channel, cell phones 1600 share several channels. The band allocated to TDMA is broken into frequency channels, each with a carrier frequency and a reasonable separation between channels. Thus user channels for the most common implementations of TDMA can be represented as a two-dimensional array, in which the rows represent frequency channels and the columns represent time slices.

CDMA is based on vector arithmetic. In a sense, CDMA performs inter-cell-phone coordination using data flow. Instead of breaking up the band into frequency channels and time slicing these, CDMA regards the entire band as an n-dimensional vector space. Each channel is a code that represents a basis vector in this space. Bits in the signal are represented as either 1 or –1, and the modulation is the inner product of this signal and a basis vector of mobile unit 1506 or cell phone 1600. This process is called spreading, since it effectively takes a narrowband signal and converts it into a broadband signal.

Demultiplexing is simply a matter of taking the dot-product of the received signal with the appropriate basis vector, obtaining the original 1 or –1. With fast computation and the appropriate codes or basis vectors, the signal can be modulated without a carrier frequency. If this is not the case, a carrier and analog techniques can be used to fill in where computation fails. If a carrier is used, however, all units use the same carrier in all cells.

Fig. 22 shows TDMA and CDMA signals for four cell phones 1600. With reference to Fig. 22, for TDMA, each cell phone 1600 is assigned a time slice during which it can transmit. Cell phone 1 is assigned time slice t0, cell phone 2 is assigned

time slice t1, cell phone 3 is assigned time slice t2, and cell phone 4 is assigned time slice t3. For CDMA, each cell phone 1600 is assigned a basis vector that it multiplies with its signal. Cell phone 1 is assigned the vector:

$$\begin{pmatrix} -1 \\ 1 \\ -1 \\ 1 \end{pmatrix}$$

Cell phone 2 is assigned the vector:

$$\begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \end{pmatrix}$$

Cell phone 3 is assigned the vector:

$$\begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}$$

Cell phone 4 is assigned the vector:

$$\begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix}$$

Notice that these vectors form an orthogonal basis.

B. Handheld Web Browser

In the previous subsection, we demonstrated our methodology on a cell phone with a top-down design approach. In this subsection, we demonstrate our methodology with a bottom-up approach in building a handheld Web browser.

Fig. 23A is a LCD touch screen component 2300 for a Web browser GUI (shown in Fig. 24A) for a wireless device 1506. With reference to Fig. 23A, a LCD touch screen component 2300, has an LCD screen 2302 and a touch pad 2304.

Fig. 23B is a Web page access component 2350 for fetching and formatting web pages. With reference to Fig. 23B, web access component 2350 has a page fetch subcomponent 2352 and a page format subcomponent 2354. Web access component 2350 reads hypertext markup language (HTML) from a connection interface 2356, sends word placement requests to a display interface 2358, and sends image requests to the connection interface 2356. Web access component 2350 also has a character input interface to allow users to enter page requests directly and to fill out forms on pages that have forms.

Fig. 24A shows a completed handheld Web browser GUI 2400. With reference to Fig. 24A, handheld Web browser GUI 2400, has LCD touch screen component 2300, web access component 2350, and a pen stroke recognition component 2402 that translates pen strokes entered on touch pad 2304 into characters.

Fig. 24B shows the complete component view of a handheld Web browser 2450. With reference to Fig. 24B, handheld Web browser 2450 is formed by connecting handheld Web browser GUI 2400 to connection component 1610 of cell phone 1600 (described with reference to Fig. 16) with bi-directional data transfer coordinator 604 (described with reference to Fig. 6B). Handheld Web browser 2450 is an example of mobile unit 1506, and connects to the Internet through the cellular infrastructure described above. However, handheld Web browser 2450 has different access requirements than does cell phone 1600. For handheld Web browser 2450, reliability is more important than real-time delivery. Dropped packets usually require retransmission, so it is better to deliver a packet late than to drop it. Real-time issues primarily affect download time and are therefore secondary. Despite this, handheld Web browser 2450 must coordinate media access with cell phones 1600, and so it must use the same protocol as cell phones 1600 to connect to the network. For that reason, handheld Web browser 2450 can reuse connection component 1610 from cell phone 1600.

It will be obvious to those having skill in the art that many changes may be made to the details of the above-described embodiments of this invention without departing from the underlying principles thereof. The scope of the present invention should, therefore, be determined only by the following claims.